Avaliação das Técnicas de Compactação Aplicáveis a Programas em Linguagens Funcionais em Arquiteturas Superescalares

Mariza A. S. Bigonha[1]
Patrícia Campos Costa [2]
Roberto S. Bigonha [3]

Abstract

The need for fasters processors and efficient executions has motivated the designers of new architectures and methods for code otimization. This paper presents important aspects of the superscalar architectures, functional languages and the compaction methods used with imperative languages programs. It also presents a compaction method known as pessimistic algorithms based on software pipelining. The method makes a global compaction of the program and can be applied to programs written in functional languages, can be extended to the compaction of general recursive functions and can be used with superscalar machines.

Resumo

O desejo de obtenção de processadores e execuções cada vez mais eficientes tem motivado a criação de novas arquiteturas e métodos de otimização de código. Este artigo apresenta uma revisão da bibliografia, salientando aspectos importantes das arquiteturas superescalares e das linguagens funcionais. Mostra as vantagens e desvantagens relativas aos métodos de compactação existentes para linguagens imperativas. O artigo apresenta também um método de compactação conhecido por algoritmos pessimistas que é baseado na filosofia de **Software Pipelining**. Esse método realiza uma compactação global do programa e pode ser aplicado para programas escritos em linguagens funcionais, pode ser estendido para a compactação de funções recursivas em geral e é aplicável em arquiteturas superescalares.

[1]DSc (PUC/RJ - 1994). Departament of Computer Science, Federal University of Minas Gerais, Belo Horizonte - MG - Brazil, E-mail:mariza@dcc.ufmg.br

[2]Master student at the Computer Science Departament, Federal University of Minas Gerais, Belo Horizonte - MG - Brazil, E-mail: pccosta@dcc.ufmg.br.

[3]PhD. (UCLA/USA 1981). Departament of Computer Science, Federal University of Minas Gerais, Belo Horizonte - MG - Brazil, E-mail: bigonha@dcc.ufmg.br.

1 Introdução

Programas em linguagens funcionais consistem em definições de funções e aplicações de funções. O elemento principal nas linguagens funcionais é a expressão, enquanto que nas linguagens imperativas é o comando. Não existe o conceito de variáveis do ponto de vista de que variáveis podem mudar de valor no decorrer da execução do programa. Uma característica importante deste tipo de linguagem é que uma expressão possui um valor bem definido, portanto a ordem na qual a expressão é avaliada não afeta o resultado final. Outro ponto importante das linguagens funcionais advém do fato de que dada a sua natureza, elas oferecem mais oportunidades para explorar o paralelismo e outras facilidades inerentes às arquiteturas superescalares que as linguagens imperativas.

Uma das principais características das arquiteturas superescalares é a separação dos componentes do processador em unidades funcionais e a habilidade de despachar e executar mais de uma instrução por ciclo de máquina. Estas propriedades auxiliam na construção de um compilador permitindo que o mesmo produza um bom código para estas máquinas. O paralelismo de grão fino para esta classe de arquitetura denomina-se *compactação*. A compactação consiste em reconhecer e escalonar grupos de operações que podem ser executadas em paralelo. A compactação pode ser *local*, envolvendo trechos de programas sem desvios ou pode ser *global* abrangendo as operações de desvios.

Várias técnicas de compactação foram desenvolvidas para ambientes baseados em linguagens imperativas. Para compactação global, as mais importantes são **Trace Scheduling** [11] e **Percolation Scheduling** [21]. Para a compactação de estruturas de controle, usa-se uma técnica denominada **Software Pipelining** [18]. Outras abordagens utilizadas em *loop* são: (i) desdobramento de *loops*, **Loop Unrolling** [14, 5], caracterizada por construir um *loop* com um número menor de iterações; (ii) algoritmos pessimistas [9], que efetuam a execução simbólica do programa tentando executar operações, o mais cedo possível, por meio do uso de um desdobramento de *loop* controlado.

2 Arquiteturas Superescalares

O desejo de obtenção de processadores mais velozes, fez com que diversas técnicas para exploração do paralelismo existente nos diversos níveis hierárquicos que formam um sistema de computador fossem desenvolvidas. Dois tipos de paralelismos podem ser explorados, o paralelismo de alto e baixo nível.

O paralelismo de alto nível é encontrado onde dois ou mais processadores executam um mesmo trecho de programa. Este tipo de paralelismo é basicamente limitado pelo desempenho dos processadores e pela sua rede de interconexão.

Já o paralelismo de baixo nível explora o paralelismo existente no interior de um único processador. As arquiteturas superescalares surgiram graças ao estudo das técnicas de paralelismo de baixo nível e ao avanço da tecnologia. Elas exploram o paralelismo a nível de instrução, e se caracterizam pela presença de muitas unidades funcionais que

podem operar em paralelo possibilitando a execução de mais de uma instrução por ciclo de máquina. Normalmente elas fazem, também, uso do mecanismo de *pipeline*. Uma *pipeline* pode ser definida como uma janela onde várias instruções seqüenciais executam simultaneamente, normalmente em fases distintas, e não podem depender uma da outra.

Uma configuração típica desta classe de arquiteturas possui unidades de execução de ponto fixo e ponto flutuante independentes, muito embora também seja possível outras unidades mais especializadas. Como exemplos de arquiteturas superescalares têm-se as arquiteturas MC88100/MC88200 [20], que possuem quatro unidades funcionais distintas: para operações com inteiros, ponto flutuante, para instruções e dados. Outro exemplo, o RS/6000 da IBM [22] possui, além das duas unidades citadas anteriormente, uma unidade de desvio. Todas as arquiteturas executam a maioria das instruções em um único ciclo de máquina e fazem uso intensivo de *pipeline*.

Nas arquiteturas superescalares, dois tipos de paralelismos podem ser explorados: (i) o primeiro é relacionado com instruções do mesmo tipo que são executadas simultaneamente em unidades funcionais iguais ou pela utilização de *pipelining* em cada unidade funcional; (ii) o segundo é relacionado com instruções de tipos diferentes que podem ser executadas nas diferentes unidades funcionais.

3 Linguagens Funcionais

Programar em linguagens funcionais consiste em construir definições e expressões que serão avaliadas pelo computador. As expressões são as unidades principais da linguagem, elas são reduzidas, ou seja, transformadas de acordo com as definições, até que se chegue em uma forma normal (se existir). As linguagens funcionais são muito simples, concisas, flexíveis e poderosas. Suas características mais importantes são: (i) a ausência de variáveis ou efeitos colaterais; (ii)um programa é a uma função do ponto de vista matemático; (iii) a operação básica é aplicação de função; (iv) o programa é aplicado à entrada e o valor resultante é a saída do programa; (v) o valor de uma expressão depende apenas do seu contexto textual, não da computação histórica, portanto, não existe conceitos de estado, contador de programa ou armazenamento.

Um programa funcional tem como representação natural uma árvore ou grafo. A avaliação se processa por meio de passos simples que podem ser feitos aplicando-se reduções no grafo. A Figura 1 ilustra este processo. Dada a função: `f x = (x+2 ) (x−4)` em Miranda [26], que é uma linguagem funcional, a mesma pode ser representada como em Figura 1 (a). A aplicação de função `f 4` em Figura 1 (b), por exemplo, resulta na árvore da Figura 1 (c). Executando a adição ou a subtração, em qualquer ordem, temos (d), e executando a multiplicação resulta em 0.

As linguagens funcionais existentes são bem parecidas diferindo-se mais na sintaxe do que na semântica. Exemplos de linguagens funcionais são: SASL [25], ML [13], KRC [27], Hope [6], LML [4], Miranda [26], Orwell [28] e Script [7].

Linguagens imperativas, como Fortran e Pascal, foram implementadas para serem usadas

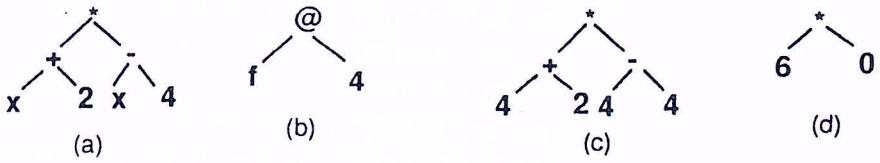

Figura 1: Representação de aplicações de função em Miranda em grafos

em desenvolvimentos de programas nas arquiteturas de Von Neumann. Estas máquinas foram projetadas para executar operações seqüenciais em itens de dados escalares. Como conseqüência estas linguagens reforçam uma seqüência artificial na especificação de algoritmos. Essa seqüência não só adiciona verbosidade aos algoritmos como pode dificultar a execução do algoritmo em processadores superescalares ou arquiteturas paralelas.

As linguagens funcionais foram desenvolvidas com o propósito de permitir que um algoritmo seja capturado no programa, eliminando qualquer necessidade de inserção de detalhes não relacionados com o problema, evidenciando a correção do programa com uma análise mínima, sendo ao mesmo tempo independentes da máquina e de alto nível. Portanto, os programas escritos em linguagens funcionais são de mais alto nível do que aqueles escritos em linguagens imperativas.

As linguagens funcionais podem, ainda, ser caracterizadas como linguagens de redução visto que a execução de programas funcionais consiste em reduzir as aplicações de funções aos seus resultados. Duas abordagens podem ser adotadas: redução de *strings* ou redução de grafos. Na redução de *strings* cada aplicação de função é avaliada e seu resultado é substituído no string que é a representação da expressão. As expressões não são compartilhadas e em cada ocorrência da expressão, a mesma redução deve ser feita. Na redução de grafos, os resultados das aplicações das funções são substituídos no grafo e é possível ter referências para as expressões já calculadas anteriormente evitando uma nova avaliação, contudo aqueles nodos que deram origem às transformações efetuadas e que não são mais necessários, precisam ser retirados por meio de um mecanismo de coleta de lixo (*garbage collection*). Uma análise das duas abordagens em relação ao tempo de execução e ao espaço gasto nos diria que, na redução de *strings*, temos a utilização de menos espaço e mais execução e na redução de grafos, utilizamos mais espaço e menos execução.

Outro fator importante na determinação da eficiência da execução de programas funcionais é a escolha da próxima aplicação a ser reduzida. Há duas possibilidades: mais interna ou mais externa. Na redução da aplicação mais externa, uma instrução é executada somente quando seu resultado é necessário para uma outra redução, sendo assim mais compatível com a abordagem redução de grafos. Por outro lado, uma redução mais interna, é mais compatível com a técnica de redução por *strings* já que uma instrução só é executada quando todos os seus argumentos já tiverem sido avaliados.

Levando em consideração os fatos expostos nos parágrafos anteriores, a escolha da técnica que propicia uma execução mais eficiente pode variar de acordo com as características da arquitetura utilizada e da classe de programas a serem executados. Uma das principais características das arquiteturas superescalares é separação dos componentes do processa-

dor em unidades funcionais e a habilidade de despachar e executar mais de uma instrução por ciclo de máquina. Se as diferentes unidades funcionais de uma máquina nesta família de arquiteturas tiverem mais de uma unidade com a mesma função, a redução por *strings* pode ser mais eficiente, pois várias aplicações de funções poderiam ser executadas ao mesmo tempo em unidades funcionais idênticas e substituídas nos *strings* sanando assim, o problema da execução excessiva que este tipo de redução apresenta. A redução de grafos também poderia tirar proveito das unidades funcionais idênticas executando em paralelo as operações replicadas evitando a inserção de novos ponteiros no grafo. Contudo seria indispensável, a presença de um mecanismo que facilitasse a operação de coleta de lixo na arquitetura para tornar a técnica de redução de grafos mais eficiente. Levando em conta que as arquiteturas superescalares, em geral, não possuem uma replicação de unidades funcionais idênticas e que estas arquiteturas não possuem mecanismos que facilitem coleta de lixo, então, não podemos afirmar qual dos dois métodos é mais eficiente para esta classe de arquiteturas, a escolha de um deles vai depender das características de cada arquitetura isoladamente. Podemos, apenas, concluir que um fator importante para o aumento da eficiência de execução da técnica de redução de *strings* em uma arquitetura superescalar depende do número de unidades funcionais idênticas enquanto que a redução de grafos exige, também, a presença de um mecanismo de coleta de lixo.

4 Técnicas de Compactação

As técnicas de compactação usadas para explorar o paralelismo de baixo nível nas arquiteturas superescalares podem ser desenvolvidas em hardware ou software e se classificam em global e local. Para que se faça um bom uso dos recursos disponíveis na arquitetura é necessária a construção do grafo de dependências, a descrição dos recursos da máquina e empacotamento das instruções paralelas.

A compactação local se limita a blocos básicos. Um bloco básico é um trecho de programa que não possui pontos de entrada, só se for o primeiro e nem pontos de saída, só se for o último, ou seja, não existem instruções de desvios condicional ou incondicional em seu interior. Este tipo de compactação não explora o paralelismo existente entre os blocos básicos. Após a divisão do programa em blocos básicos, os algoritmos propostos para este tipo de compactação [10, 17, 23, 19] tentam empacotar, agrupar, as instruções a serem despachadas, verificando a possibilidade da execução de comandos de um mesmo bloco básico paralelamente. Como tudo é feito dentro do contexto do bloco básico, usando somente a compactação local é possível perder oportunidades de mover instruções de um bloco para outro, o que traria um ganho significante. Contudo, a compactação local pode ser usada juntamente com a compactação global para se obter melhores resultados. Outras técnicas desenvolvidas visando a compactação local, podem ser encontradas em [8, 15] e [16]. Neste texto nos focalizaremos na compactação global.

A compactação global explora o paralelismo do programa todo, abrange os desvios e pode aumentar a taxa de concorrência transferindo instruções de um bloco básico para outro. Este método consiste na construção do grafo de fluxo de controle, sua análise e movimentação das instruções obedecendo os critérios de dependências estabelecidas no grafo.

Nas próximas seções serão apresentados alguns métodos desta abordagem. Eles estão divididos em duas categorias: os métodos de compactação global (**Trace Scheduling** [11], **Percolation Scheduling** [21]), e os métodos para compactação de estruturas de controle (**Software Pipelining** [18], **Loop Unrolling** [14, 5]). Apresentamos também um método, denominado **Algoritmos Pessimistas**, que se encaixa nas duas categorias e que mostra a aplicabilidade de algoritmos desta natureza em programas escritos em linguagens funcionais.

4.1 Trace Scheduling

Esta é uma técnica de compactação global, que explora o paralelismo de trechos de programas denominados *traces*[11]. Ele funciona da seguinte forma: inicialmente constrói-se o grafo, DAG (Directed Acyclic Graph) para a análise de dependência das instruções do programa a ser compactado. A partir do DAG construido o **Trace Scheduling**, basicamente, executa em 4 fases: escolha do trecho, pré-processamento do trecho, escalonamento das instruções do *trace* e pós-processamento do trecho.

4.2 Percolation Scheduling

Esta técnica [21] foi desenvolvida na tentativa de acabar com algumas das deficiências da técnica **Trace Scheduling** [11], entre elas: (i) a compactação de somente trechos isolados do programa, que faz com que o paralelismo não seja totalmente explorado; (ii) o custo das inserções de blocos de reparo; (iii) a compactação de somente um trace de cada vez; (iv) a reduzida interação do usuário com o processo de compactação; (v) a falta de evidência da eficiência para aplicações científicas.

Para superar estas deficiências, foi desenvolvido um ambiente interativo para a geração do código paralelo, reorganizando-se globalmente os programas de aplicação, visto que a movimentação de comandos é feita além das fronteiras dos blocos básicos.

O objetivo do algoritmo de **Percolation Scheduling** [21] é maximizar o paralelismo movendo operações de nodo para nodo de modo a maximizar o número de operações paralelas no DAG final. O ambiente provê primitivas de transformações que fazem alterações no DAG de acordo com as dependências presentes, bem como primitivas que realizam operações inversas, preservando a equivalência semântica do programa. Estas primitivas são:

Delete: elimina um vértice do grafo. Arestas dos antecessores do vértice removido são dirigidas para seu sucessor para neutralizar o efeito da operação.
Move_op: movimenta um comando para o vértice adjacente anterior.
Move_cj: movimenta um comando de desvio condicional para um antecessor.
Unification: unifica diversos comando idênticos.

Utilizando-se estas primitivas, a ordem dos comandos do programa a ser compactado pode ser modificada, preservando-se a equivalência semântica em todos os caminhos do grafo que são afetados pelas transformações. Aplicações sucessivas das primitivas migram os comandos em direção ao topo do grafo que representa o programa, permitindo o empacotamento dos comandos do programa. Estas primitivas têm sido usadas como valiosas ferramentas em diversos experimentos. Este método se encaixa muito bem na execução de linguagens funcionais através de redução de grafos, pois uma vez construido o grafo a obtenção do DAG é bem natural.

4.3 Loop Unrolling

A técnica de Loop Unrolling [14, 5] consiste na construção de um *loop* com um menor número de iterações, repetindo-se o corpo do *loop* original seqüencialmente. Com este esquema temos um maior número de operações dentro do *loop*, eliminando ciclos de código extra dentro do *loop* original. Como conseqüência, as instruções do *loop* podem ser escalonadas de modo a maximizar o uso das unidades funcionais diponíveis, aumentando o paralelismo de instruções. As desvantagens desta técnica são: o tamanho do código gerado é maior e é necessário mais registradores para armazenar resultados intermediários. A Figura 2, extraída de [5], mostra a melhora do desempenho com o desdobramento do *loop* duas vezes. Em (b) a sobrecarga do *loop* é reduzida à metade, pois duas iterações são executadas antes do teste e o paralelismo de instruções é aumentado, pois a segunda atribuição pode ser executada enquanto os resultados da primeira estão sendo armazenados e as variáveis do *loop* estão sendo atualizadas.

```
do i=2, n-1
    a[i] = a[i] + a[i-1] * a[i+1]
end do
```

(a)

```
do i=2, n-2, 2
      a[i] = a[i] + a[i-1] * a[i+1]
      a[i+1] = a[i+1] + a[i] * a[i+2]
end do
if (mod(n-2,2) = 1) then
      a[n-1] = a[n-1] + a[n-2] * a[n]
end if
```

(b)

Figura 2: Loop Unrolling : (a) *loop* original , (b) *loop* desenrolado duas vezes.

4.4 Software Pipelining

O algoritmo proposto por Lam[18] utiliza uma técnica de redução hierárquica combinada ao **Software Pipelining** possibilitando a aplicação do **Software Pipelining** aos loops mais internos inclusive aqueles com desvios condicionais. A abordagem proposta escalona o programa hierarquicamente começando com as estruturas de controle mais internas. Assim que cada construção é escalonada, a construção é toda substituída por um nodo. O processo estará completo quando o programa estiver reduzido a um simples nodo. Esta técnica se aplica muito bem ao método de execução de linguagens funcionais se

considerarmos a redução de grafos, pois na redução de grafos, também as funções vão sendo executadas e substituidas no grafo até que se chegue a um simples nodo de forma análoga ao **Software Pipelining**.

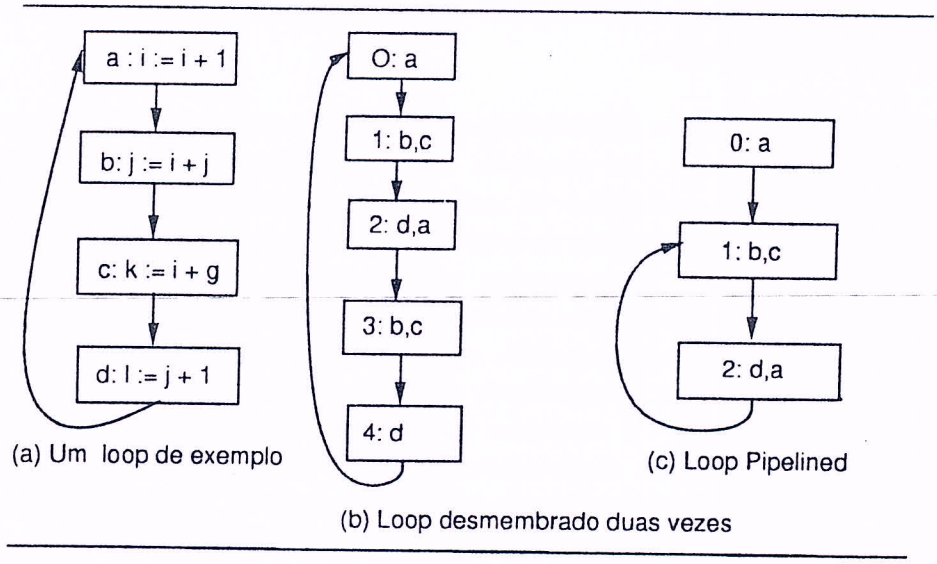

Figura 3: Loop Unrolling e Software Pipeline

O objetivo da técnica **Software Pipelining** [18] é tentar reduzir o atraso provocado por acessos à memória para a obtenção dos operandos necessários à execução de uma operação em uma unidade funcional. Ela faz a transferência das instruções de acesso à memória para iterações anteriores ao *loop* de modo que os operandos estejam diponíveis em registradores no momento em que a operação for executada. As operações de uma iteração do *loop* são quebradas em s estágios e uma única iteração executa o estágio 1 da iteração `i`, o estágio 2 da iteração `i-1`, e assim por diante. A técnica reduz o intervalo entre iterações consecutivas do *loop*, diminuindo seu tempo total de execução. Ela reduz o custo do *startup* de cada iteração do *loop*, apesar de necessitar de código de inicialização do *loop* e código de finalização depois do *loop* para drenar o *pipeline* para as últimas iterações.

As técnicas de *software pipelining* apresentam a possibilidade de produzir um código melhor com tempo de compilação menor que outras técnicas de escalonamento. Por exemplo a Figura 3, extraída de [1], ilustra este fato, comparando-a com Loop Unrolling [14, 5].

Na Figura 3(a) temos o *loop* original, em (b), é mostrado o resultado da aplicação da técnica **Loop Unrolling**, Seção 4.3 (a), onde o *loop* foi desmembrado duas vezes. Em (c) é apresentado o resultado da aplicação de **Software Pipelining**. Observando o item (c) percebe-se que este *loop* possui dois tipos de paralelismos: um paralelismo dentro do corpo do *loop*, onde b e c podem ser executados simultaneamente, e um paralelismo entre as iterações do *loop*, onde o d de uma iteração pode sobrepor a da próxima iteração. A abordagem de **Loop Unrolling** usada em (b) permite que o paralelismo seja explorado entre algumas iterações do *loop* original, mas existe uma seqüência imposta entre as

iterações do corpo do *loop* desmembrado. Se o *loop* tivesse sido totalmente desmembrado, todo o paralelismo interno ao *loop* e entre as iterações do *loop* poderia ter sido explorado, entretanto o desmembramento total é em geral impossível ou impraticável em determinados casos onde o tamanho do código cresceria muito. Já em (c) pode ser observado que o algoritmo de **Software Pipelining** provê uma maneira direta de explorar o paralelismo presente dentro do *loop* e entre suas iterações de tal forma que o mesmo atinge o efeito de um completo desmembramento.

Além da abordagem de **Software Pipelining** apresentada, existem na literatura várias outras implementações possíveis, por exemplo: Aiken [1] apresenta um algoritmo para o **Software Pipelining** que contabiliza as restrições dos recursos de máquina de uma maneira suave integrando a gerência de restrição de recursos com o software pipelining. O algoritmo é formado por dois componentes: um escalonador e um analisador de dependências. O escalonador é usado para construir um *loop* paralelizado a partir de um *loop* seqüencial. Para cada instrução paralela, o escalonador seleciona operações a serem escalonadas baseado no conjunto de operações disponíveis para escalonamento naquela instrução e recursos disponíveis. O conjunto de operações disponíveis é mantido pelo analisador de dependências globais. Após a construção do *loop* paralelizado, o algoritmo de *software pipelining* verifica por estados que podem ser *pipeline*. Wang[29] propõe uma técnica de *software pipelining* chamada **Decomposed Software Pipelining**, onde o problema das dependências de dados e restrição de recursos são divididos em dois subproblemas: (i) livrar o *loop* das depêndencias de dados cíclicos e (ii) livrá-lo da restrição de recursos. Os problemas podem ser tratados em qualquer ordem e experimentos preliminares mostraram que independente da ordem escolhida, os resultados são eficientes e levam a um melhor resultado do que as abordagens existentes tanto em complexidade de tempo quanto em eficiência de espaço com uma menor complexidade de computação [29].

Aiken [2] utiliza uma abordagem onde o programa é representado por seu grafo de dependências. O algoritmo examina uma execução parcial de um *loop*, por exemplo as primeiras i iterações, e tenta escalonar estas i iterações o mais cedo possível. As dependência entre *loops* são tratadas da seguinte forma: se a maior cadeia de dependências onde a instrução x depende tem comprimento j então x é escalonado no tempo j.

Esta abordagem tenta se aproveitar do fato de que, como as dependências de um *loop* apresentam certas regularidades especificadas pelo grafo de depêndencias, então ao escalonar uma grande porção da execução de um *loop*, algum comportamento repetitivo pode ser revelado. Este fato pode ser usado para obter um escalonamento ótimo para o *loop*.

4.5 Algoritmos Pessimistas

Pouzet [24] apresenta um método de compactação que é uma generalização do **Percolation Scheduling** [21] e **Perfect Pipelining** [2] e mostra que também é possível aplicar os métodos de compactação usadas em linguagens imperativas aos programas escritos em linguagens funcionais. O método proposto, também denominado algoritmo pessimista [9] se caracteriza pela realização de uma execução simbólica de um programa tentando executar as operações o mais cedo possível. Para isto eles utilizam um desmembramento

controlado de *loops* até que um padrão repetido ocorra. O programa, nesta abordagem, é representado por um árvore e toda vez que é detectado um nodo pronto, os mesmos são movimentados para cima. Um nodo é dito estar pronto se todos os seus argumentos já foram computados. Ao encontrar um nodo pronto que é equivalente a um já compactado, ele é substituido por uma chamada para uma nova função que executa os mesmos comandos do nodo pronto. Em outras palavras, o processo de compactação detecta nodos prontos e os passa para cima para que sejam despachados o mais rápido possível fazendo as transformações no programa, como no método **Percolation Scheduling** apresentado na Seção 4.2 (b). Quando uma chamada recursiva é detectada é feito um desdobramento, como na técnica de **Loop Unrolling**. E assim sucessivamente até que a condição de parada seja atingida. A condição de parada é a detecção de um nodo pronto igual a outro nodo pronto avaliado anteriormente. Neste momento a chamada para a função recursiva é substituída por uma outra chamada correspondente. Uma função é recursiva quando ela invoca a si mesma direta ou indiretamente. O método proposto por Pouzet [24] pode ser melhor entendido observando seu exemplo. Neste exemplo, suponha um trecho de programa em uma linguagem funcional que define uma função `f` que recebe como argumentos uma lista `x` e um valor escalar `accu`. A função `f` executa as primitivas `f1` e `f2` e chama a si mesmo. As barras horizontais indicam os nodos prontos, o par `[ ]` representa uma lista vazia, `hd` representa a cabeça da lista e `tl` representa a cauda da lista.

$$\begin{array}{lll} \text{let rec f} = \lambda \text{x, accu,} & \text{if x} = [\,] & \\ & \text{then accu} & \\ & \text{else } \overline{f(\overline{tl(x)}, \text{f2}(\text{f1}(\overline{hd(x)}), \text{accu}))} & \end{array} \qquad \textbf{(a)}$$

O teste if `x = [ ]`, e as funções *hd* e *tl* são nodos prontos, portanto devem ser movimentados para cima. O teste não é movimentado pois já se encontra na posição correta. Após a movimentação das funções `hd` e `tl` obtém-se a seguinte configuração:

$$\begin{array}{lll} \text{let rec f} = \lambda \text{x, accu,} & \text{if x} = & [\,] \\ & \text{then} & \text{accu} \\ & \text{else} & \text{let x1} = \text{tl(x)} \\ & & \text{x2} = \text{hd(x)} \\ & & \text{in } \overline{f(x1, f2(f1(x2).accu))} \end{array} \qquad \textbf{(b)}$$

Agora o próximo passo é substituir a chamada recursiva de `f` pela sua definição obtendo:

```
let rec f = λ x, accu. if x = [ ]
                       then accu
                       else let x1 = tl ( x )
                                x2 = hd( x )
                            ____
                            (λ x, accu. if x = [ ]
                                        then accu
                                        else let x1 = tl ( x )
                                                 x2 = hd(x)
                                             in f(x1, f2( f1(x2), accu))
                            )
                       in                                          (x1,f2(f1(x2),accu))
```

(c)

Agora, movimentando os nodos prontos para cima, `x = [ ]` e `f1(x2)`, e renomeando o primeiro $\hat{x}$ do trecho entre parêntesis para `x5` e temos:

```
let rec f = λ x,  accu. if x = [ ]
                        then accu
                        else let x1 = tl(x)
                                 x2 = hd(x)
                             in let x3 = f1(x2)
                                in if x1 = [ ]
                                   then (λ accu. accu) (f2 ( x3, accu))
                                   else (λ accu. let x5 = tl( x1)
                                                 x2 = hd( x1 )
                                         in f(x5, f2( f1(x2), accu))
                                         )                          (f2(x3,accu))
```

(d)

Repetindo o processo de mover os nodos prontos e fazendo a redução de `(λ accu.accu)(f2 (x3, accu))` para `f2( x3, accu)`, chegamos a:

```
let rec f = λ x, accu. if x = [ ]
                       then accu
                       else let x1 = tl (x)
                                x2 = hd(x)
                            in let x3 = f1(x2)
                               in if x1 = [ ]
                                  then f2(x3, accu)
                                  else let x4 = f2(x3, accu)
                                           x5 = tl(x1)
                                           x2 = hd(x1)
                                       in λ acc. ( f (x5 , f2 ( f1 (x2), accu)) (x4)
```

(e)

O termo `λ accu.f( x5, f2(f1(x2),accu))(x4)` pode ser reduzido a `f(x5,f2(f1(x2),x4))` que já está presente na letra (b), então com esta redução chegamos ao fim da ação e o código resultante define uma nova função `ff`. Substituindo `f(x5, f2(f1(x2),x4))` por uma chamada para esta função obtem-se o código final que não pode mais ser compactado:

No exemplo mostrado, a função `f` é *tail-recursive* (Veja Seção 4.6) e os argumentos são colocados nos registradores assim que possível. Renomeação e operações de movimentação são condicionadas pelos recursos disponíveis. Nos laços em geral, como foi visto nas Seções

```
                                                  let rec ff = λ x2, x1, accu. let x3 = f1(x2)
                                                                              in if x1 = [ ]                (f)
let rec f = λ x, accu. if x = [ ]                                               then f2 (x3, accu)
                       then accu                                                else let x4 = f2(x3, accu)
                       else let x1 = tl(x)                                           x5 = tl(x1)
                                x2 = hd(x)                                           x2 = hd(x1)
                            in ff (x2, x1, accu)                                in ff ( x2, x5, x4)
```

4.3 e 4.4, a linearização tem que ser controlada e a estratégia onde um pedaço do código é linearizado não converge. O mesmo acontece com funções que possuem mais de uma chamada recursiva. Pouzet [24] propõe como controle limitar o conjunto de nodos prontos e além disso, ele propõe que quando o conjunto de nodos prontos selecionados de um termo é incluido em um subtermo que já foi compactado, o subtermo é substituido por uma chamada para a nova função definida. Desta forma, o processo de compactação proposto por ele é capaz de compactar, também, funções sem recursividade de cauda.

4.6 Análise das Técnicas de Compactação

Fazendo uma análise das duas abordagens examinadas para a compactação global, chegamos à seguinte conclusão: a técnica **Trace Scheduling**[11] é vantajosa, pois é uma técnica de compactação global, que explora paralelismo além dos blocos básicos. Entretanto, apesar de poder ser usada em qualquer aplicação, foi demonstrado na literatura que é mais eficiente para aplicações científicas [12]. Outras desvantagens são: a interação com o usuário não é satisfatória, o paralelismo nos limites dos traces pode deixar de ser explorado e o código de reparo incluído nos pontos de entrada e saída do trace compactado possui um certo custo.

A técnica **Percolation Scheduling** [21] foi desenvolvida com o objetivo de superar as deficiências de `Trace Scheduling` [11] citadas acima e, portanto, apresenta uma maior interação com usuário. Explora o paralelismo além dos traces e não é necessária a inclusão de código de reparo incluidos nos pontos de entrada e saída do trace compactado. Contudo, a aplicação de **Percolation Scheduling**[21] pode ser muito trabalhosa e os rearranjos do grafo resultante das aplicações das primitivas também acarretam um certo custo. Uma forma de obter melhores resultados seria usá-la em conjunto com a técnica **Trace Scheduling**[11].

Em relação às duas abordagens de compactação de *loops* estudadas, verificou-se que a combinação da técnica *loop unrolling* com *software pipelining* [18] pode produzir um melhor resultado, pois **Loop Unrolling** reduz o código extra, enquanto que o **Software Pipelining** [18] reduz o custo de startup de cada iteração. A Figura 3 da Seção 4.4 ilustrou este fato por meio de um exemplo. Mas cuidados especiais devem ser tomados porque embora algumas vezes, o uso de **Loop Unrolling** juntamente com **Software Pipelining** pode aumentar a eficiência do tempo de execução, esta combinação pode degradar a eficiência em termos de espaço, se o *loop* for desmembrado um número excessivo de vezes.

Outra consideração importante diz respeito às transformações de funções com recursividade de cauda em laços. Uma função é tail-recursive, ou com recursividade de cauda, se a última ação da função é chamar a si mesma e retornar o valor da chamada recursiva sem executar qualquer outro comando, isto é, não há computações a serem feitas depois do retorno da chamada recursiva. Este tipo de função pode ser eliminado, traduzindo-a em um laço pela substituição da chamada da função por um goto para o seu início. A Figura 4, extraída de [5] ilustra a eliminação da recursividade de cauda e mostra um exemplo de função sem recursividade de cauda.

```
recursive logical function inarray(a, x, i, n)
   real x, a[n]
   integer I, n

   if (i > n) then inarray = .FALSE.
   else if (a[i] = x) then
        inarray = .TRUE.
   else inarray = inarray(a, x, i+1, n)
   end if
   return
```

(a) Procedimento com recursividade de cauda

```
recursive logical function inarray(a, x, i, n)
     real x, a[n]
     integer I, n

1    if (i > n) then inarray = .FALSE.
     else if (a[i] = x) then
          inarray = .TRUE.
     else i = i + 1
     goto 1
     end if
     return
```

(b) Depois da eliminação da recursividade

```
recursive integer function sumarray(a, x, i, n)
     real x, a[n]
     integer I, n

     if (i = n) then sumarray = a[i]
     else sumarray = a[i] + sumarray(a, x, I=1, n)
     end if
     return
```

(c) Um procedimento *non-tail recursive*

Figura 4: Eliminação de Recursividade de Calda

Dado este fato, uma generalização natural do princípio de software pipelining [18] pode ser utilizada para funções recursivas visto que a recursividade das funções *tail-recursive* pode ser eliminada como mostrado.

5 Conclusão

As arquiteturas superescalares foram desenvolvidas graças ao estudo de técnicas de exploração do paralelismo existente no interior de um único processador. Uma das principais

características das arquiteturas superescalares é a existência de várias unidades funcionais que podem operar em paralelo. A exploração do paralelismo de baixo nível nestas arquiteturas se denomina compactação.

A compactação para as arquiteturas superescalares consiste em reconhecer grupos de instruções elementares para serem escalonadas em paralelo e observando-se dois fatores importantes: a máxima utilização das unidades funcionais e a equivalência semântica do programa. O compilador extrai o paralelismo dos programas a partir de uma análise de dependência de suas instruções. Para que um método de compactação se torne utilizável dentro de todos os compiladores das arquiteturas superescalares é essencial dispor de uma implementação eficiente tanto para a compactação global como para a compactação de *loops* independente do paradigma da linguagem usada.

As linguagens funcionais apresentam várias características que oferecem mais oportunidades de explorar o paralelismo e facilidades inerentes as arquiteturas superescalares do que as linguagens imperativas. Por exemplo, o paralelismo nas linguagens funcionais é natural. O não-determinismo no momento da avaliação é natural. Pela própria natureza do DAG, se houver n unidades funcionais ou n processadores nas arquiteturas paralelas, cada ramo do DAG ou árvore pode ser disparado em paralelo. Portanto em linguagens funcionais aproveita-se melhor o paralelismo já que não existe o conceito de atualização de variáveis nestas linguagens e nem efeito colateral. Em contrapartida existem também alguns fatores que podem degradar a execução de programas funcionais, um deles seria a inexistência de uma arquitetura funcional. Na verdade este é um problema fundamental das linguagens funcionais, porque as máquinas foram construídas para as linguagens imperativas onde existe a noção de atualização de memória o que não é natural em uma linguagem funcional. As arquiteturas de Von Neumann não são adequadas para a execução de programas em linguagens funcionais. Mas como não existe uma máquina funcional, uma forma de compactar programas neste novo paradigma é traduzí-la para linguagens imperativas e então aplicar as transformações no grafo de controle de fluxo obtido, o que não é uma boa tática já que assim as propriedades das linguagens funcionais propícias para a compactação desapareceriam na tradução.

No estudo das técnicas de compactação foi possível observar que as técnicas de compactação global oferecem mais oportunidades de explorar o paralelismo do que a compactação local pois exploram o paralelismo existente além dos blocos básicos e que o tratamento dos laços é extremamente importante para que seja alcançado um bom nível de eficiência. Podemos concluir, também, que uma combinação das técnicas apresentadas neste trabalho é possível para aumentar a eficiência. Entretanto, interesses e propósitos devem ser avaliados para que a escolha de combinações de algoritmos seja adequada. Não existe uma técnica melhor do que a outra para todas as classes de programas. O que ocorre é que cada técnica é adequada para um determinado tipo de programa. Por exemplo, a Figura 3 ilustrou que **Software Pipelining** era mais eficiente que **Loop Unrolling** para aquele trecho de programa, entretanto em outros trechos o inverso pode ocorrer. O mesmo acontece com **Trace Scheduling** e **Percolation Scheduling**, um método pode ser mais eficiente para determinado caso, e menos eficiente para outros. Se a linguagem usada é uma linguagem funcional a questão da manutenção do grafo de dependências é mais simples porque a reconstrução deste grafo pode ser evitada devido ao fato de não haver atualizações de variáveis durante a execução de um programa funcional. As dependências

não evoluem ao longo do tempo, evitando assim o doloroso problema de modificar o grafo várias vezes. Com a construção do grafo de dependências inicial têm-se uma representação finita e estática do programa, o que facilita bastante o trabalho.

Os métodos de compactação de *loops* e global também podem ser utilizados conjuntamente. Não existe um método de compactação de laços completos que permita tomar partido de todo o paralelismo potencial de um programa, principalmente se o generalizarmos para programas recursivos. Este problema é muito complexo e um dos motivos desta complexidade consiste na ignorância durante a compilação do caminho de execução. O compilador pode compactar trechos que não serão executados ou deixar de compactar eficazmente trechos que serão executados.

A técnica apresentada na Seção 4.5 mostrou-se como uma boa solução para a compactação de linguagens funcionais nas arquiteturas superescalares apesar de ter ainda muitos pontos a serem pesquisados. Seu esquema é válido porque mostrou que é possível, fazendo algumas adaptações ou combinando técnicas propostas para programas escritos em linguagens imperativas aplicá-las em linguagens funcionais. Foi possível observar nesta técnica também a viabilidade de compactação das funções recursivas gerais. As funções *tail-recursive* já eram tratadas pelas técnicas anteriormente desenvolvidas, pois este tipo de função pode ser eliminado, transformando-a em um laço através da substituição da chamada da função por um goto para o seu início. Já as funções *non-tail recursive* não podem ser eliminadas tão facilmente. Esta técnica é capaz de compactar este tipo de função também, pois ela apresenta uma condição de parada, o que não existia nas técnicas anteriores. O controle proposto por Pouzet[24] que consiste na limitação do conjunto de nodos prontos possibilida a compactação deste tipo de função recursiva também.

Contudo, falta demonstrar se realmente existe um ganho em termos de execução. O que se fez foi simplesmente mostrar uma série de transformações aplicadas diretamente a alguns programas fontes escritos em uma linguagem funcional proposta por ele. Como continuação deste trabalho, seria interessante tentar implementar a técnica proposta para compactação em uma linguagem funcional real tendo em vista sua execução em uma máquina real e fazer um estudo de sua execução.

Referências

[1] Aiken, A. ,Nicolau, A, Novack, S. *Resource-Constrained Software Pipelining.* In IEEE Transactions on Parallel and Distributed Systems, 1996.

[2] Aiken, A., Nicolau, A. *Optimal Loop Parallelization.* In Proceedings of the SIGPLAN`88 Conference on Programming Language Design and Implementation. Atlanta, Georgia, June 22-24, 1988.

[3] Arabe,J. N. C. *Compiler Considerations and Run-Time Storage Management for a Functional Programming System.* Ph.D. Thesis. University of California, Los Angeles, 1986

[4] Augustsson, L. 1984 . *A Compiler for lazy ML. Proceedings of the ACM Symposium on Lisp anf Functional Programming,* Austin. August, pp. 218-27.

[5] Bacon, D. F., Graham, S. L., Sharp, O. J. *Compiler Transformations for High-Performance Computing.*, ACM Computers Surveys, Vol.26, No. 4, December, 1994.

[6] Burstall, R. M., MacQueen, D.B. and Sanella, D. T. 1980. *Hope: an experimental aplicative language.* CSR-62-80. Department of Computer Science, University of Edinburgh. May.

[7] Bigonha, R. S. *Script: An Object Oriented Language for Denotational Semantics (user's manual and reference).* RT 03/94, DCC, UFMG, 1994.

[8] Bradlee, David G., *Retargetable Instruction Scheduling for Pipelined Processors*, Tese de Doutorado, University of Washington, 1991.

[9] Ebcioglu, K. *A compilation technique for software pipelining of* loops *with conditional jumps.* In Procedings of the 20th Annual Workshop on Microprograming, pages 69-79, December 1987.

[10] Fisher, J. A., Ellis, J. R., Ruhenberg, J. C. and Nicolau, A. *Parallel processing: A Smart Compiler and a Dumb Machine.* In Symposium on Compiler Construction. SIGPLAN Notices, June 1984. Vol 19, Number 6.

[11] Fisher, J. A. *Trace Scheduling: A Technique for Global Microcode Compaction.* In IEEE Transactions on Computers, Vol. C-30, No. 7, July 1981.

[12] Fernandes, E. S. T., Santos, A. D. *Arquiteturas Superescalares: Detecção e Exploração de Baixo Nível.* Porto Alegre: Instituto de informática da UFRGS, 1992, 1148p.

[13] Gordon, M.J., Milner, A. J. and Wadsworth, C. P. 1979. Edinburg LCF. LNCS 78, Springer Verlag.

[14] Gasperoni, F. and Schwiegelshohn, U. *Scheduling loop on parallel processors: A simple algorithm with close to optimum performance.* In CONPAR, September 1992.

[15] Gibbons, Phillip B. and and Muchnick, Steven S., *Efficient Instruction Scheduling for a Pipelined Architecture*, Proceedings of the ACM Sigplan'86 - Symposium on Compiler Construction, 1986, July, ACM Sigplan Notices 21(7).

[16] Goodman, James R. and Wei-Chung-Hsu, *Code Scheduling and Register Allocation in Large Basic Blocks*, International Conference on Supercomputing - Conference ACM-PRESS Proceedings, 1988, St. Malo France, July.

[17] Hennessy, J. L., Gross, T. R. *Code generation and reorganization in the presence of pipeline constraints.* In Principles of Programming Languages, pages 120-127, 1982.

[18] Lam, M. S. *Software pipelining: An effective scheduling technique for VLIW machines.* In Conference on Programming Language, Design and Implementation, pages 318-328, June 22- 24 1988.

[19] Bigonha, M. A. S. *Otimização de Código em Máquinas Superescalares.* Tese de Doutorado. Departamento de Informática, PUC - RJ, Abril, 1994.

[20] Motorola, Inc., *MC88100 RISC Microprocessor User's Manual*, Prentice-Hall, Englewood Cliffs, 1990, second, New Jersey.

[21] Nicolau, A. *Percolation Scheduling: A Parallel Compilation Technique.* Technical Report. Cornell University. 1985.

[22] Bigonha, Mariza A. S., *Esquemas de Escalonamento de Instruções para a Arquitetura RISC/6000**, PUC-RJ, Departamento de Informática, 1992, Série de Monografias em Ciência da Computação, 09/92, Abril.

[23] Pingali, K. *Fine-grain compilation for pipeline machines.* Technical Report 88-934, Cornell University, August 1988.

[24] Pouzet, M. *Fine Grain Parallelisation of Functional Programs for VLIW or Superscalars Architectures.* In International conference on Applications in Parallel and Distributed Computing, April 1994.

[25] Turner, D. A. *The SASL Language Manual.* University of St Andrews. December, 1976.

[26] Turner, D. A. *Miranda - a non-stricrt functional language with polymorphic types.* In Conference on Functional Programming Languages and Computer Architecture, Nancy, pp. 1-16. Jouannaud, LNCS 201. Springer Verlag.

[27] Turner, D. A. *Recursion Equations as a Programming Language. In Functional Programming and its Applications,* Darlington et al., pp 1-28. Cambridge University Press.

[28] Wadler, P. 1985. *Introduction to Orwell.* Programming Research Group. University of Oxford.

[29] Wang, J., Eisenbeis, C. *Decomposed software pipelining: a new pproach to exploit instruction level parallelism for loop programs.* In IFIP WG 10.3 Working Conference on Architectures and Compilation Techiniques for Fine and Medium Grain Parallelism, January, 1993.